Automatic Extraction of JPF Options and Documentation

Wojciech Luks[1], Oksana Tkachuk[2], David Bushnell[2]

[1] AGH University of Science and Technology, Cracow, Poland
[2] NASA Ames Research Center, Moffett Field, CA, USA

Abstract. **Documenting existing Java PathFinder (JPF) projects or developing new extensions is a challenging task. JPF provides a platform for creating new extensions and relies on key-value properties for their configuration. Keeping track of all possible options and extension mechanisms in JPF can be difficult.**
This paper presents `jpf-autodoc-options`, a tool that automatically extracts JPF projects' options and other documentation-related information, which can greatly help both JPF users and developers of JPF extensions.

I INTRODUCTION

Java PathFinder (JPF) [1] is an open source, explicit state software model checker for Java bytecode. In addition to providing a configurable model checking engine, JPF serves as a platform for various extensions (e.g., symbolic execution and state chart model checking). To provide flexibility and a plug-in architecture, JPF has a wide range of configuration options and extension mechanisms. The configuration options allow users to easily control many aspects of the execution of existing components. The extension mechanisms make it easy to develop new plugins.

JPF relies on key-value properties for configuring its components, but currently there is no systematic way to document them. Working with such a system is difficult not only for new JPF users but also for experienced ones who work with JPF on a daily basis. This problem creates the need for a tool that can collect all options in a single place [2].

In this paper we present a tool, called `jpf-autodoc-options`, which addresses the above problems. Our tool statically analyzes existing JPF projects, extracts the information related to JPF options and extension mechanisms, and saves the extracted data using XML and wiki formats. XML is a popular and easy to parse format used for storing large collections of structured data. Wiki format has become a popular format for project documentation. More and more projects nowadays are stored in repositories like Google Project Hosting [4], where wiki format is commonly used for creating documentation. Moreover, Google Project Hosting is a planned target repository for JPF projects.

II TOOL DESCRIPTION

The `jpf-autodoc-options` tool is a JPF extension packaged as a stand-alone project (i.e., it does not require `jpf-core` in its path to run).

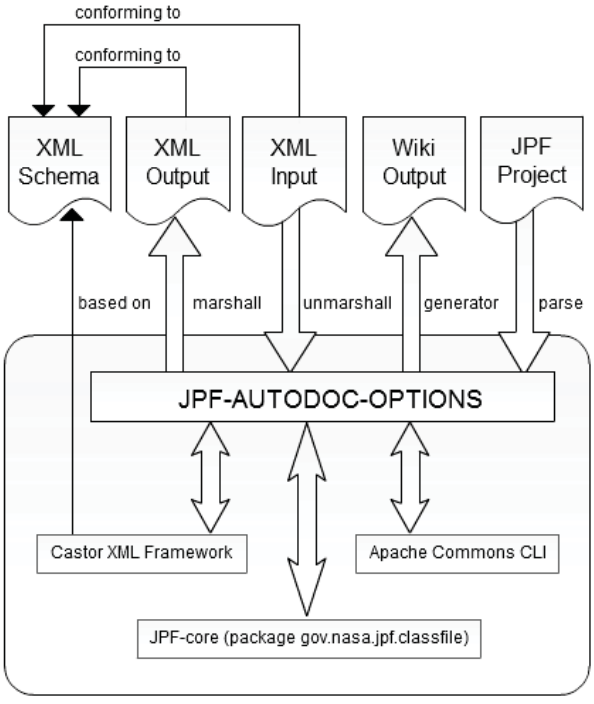

Figure 1: Tool Architecture.

Figure 1 shows the high-level architecture of the tool. The tool uses the following external libraries: Castor XML [6] for handling XML files, Apache Commons CLI [7] for user-specified (command-line) arguments, and a single `jpf-core` package called

Categ	Java Source	ByteCode (excerpts)
Option	`printInsn = config.getBoolean( "et.print_insn", false);`	`ldc @6(et.print_insn) iconst_0 invokevirtual @7("gov/nasa/jpf/Config", "get Boolean","(Ljava/lang/String;Z)Z")`
Annot	`@JPFOption(type = "Boolean", key="et.print_insn", defaultValue="true", comment="print executed bytecode instructions")`	`Lgov/nasa/jpf/annotation/JPFOption; valueCount=4 type="Boolean" key="et.print_insn" defaultValue="true" comment="print executed bytecode instructions"`
Logger	`static Logger log = JPF.getLogger( "gov.nasa.jpf .listener.IdleFilter");`	`ldc @73(gov.nasa.jpf.listener.IdleFilter) invokestatic @74("gov/nasa/jpf/JPF","getLogger", "(Ljava/lang/String;)Lgov/nasa/jpf/util/ JPFLogger;")`
CG	`vm.getSystemState(). setNextChoiceGenerator(cg);`	`aload_1 invokevirtual @241("gov/nasa/jpf/jvm/System State", "setNextChoiceGenerator", "(Lgov /nasa/jpf/jvm/ChoiceGenerator;)Z")`
DCSF	`DirectCallStackFrame frame = new DirectCallStackFrame(mainStub, 1, 0);`	`invokevirtual @207("gov/nasa/jpf/jvm/ DirectCallStackFrame","pushRef","(I)V")`

Table 1: Examples of Information Tracked by jpf-autodoc-options

`gov.nasa.jpf.classfile`. This package handles the reading of bytecode for an analyzed JPF project. The inputs to `jpf-autodoc-options` are the JPF projects to be analyzed and an XML schema used by Castor; the outputs are XML and wiki files, with information about the projects' configuration options, including potential inconsistencies and errors. The generated XML file can also be used as an input to be converted into the wiki files.

Currently, the tool supports extraction of the following types of information:

- **Options:** To configure options, JPF uses a central dictionary object `gov.nasa.jpf.Config`, which is initialized through a hierarchical set of Java property files that target three different initialization layers: site, project, application [3]. The tool tracks all `Config` usages by looking for calls of the form `config.get...(String, ...)`. Here, the name of the method describes the type of the option. The first parameter is usually the name of the option, and the rest of the parameters specify additional information, for example, the value of the option. The **Option** row in Table 1 shows an example of both Java source code and the corresponding bytecode for loading a `Boolean` option called `et.print_insn` with value `false`.

- **Option Annotations:** JPF developers are encouraged to use `@JPFOption` annotations to document JPF options. In addition to information covered by the `config.get...(String, ...)` call, the developers can add comments and document default values for the options. The **Annot** row of Table 1 shows an example of an annotation for the `et.print_insn` option. It is important to check that annotations are consistent with the code. Therefore `jpf-autodoc-options`, by default, checks for consistency between implemented options and their annotations.

- **Loggers:** To perform logging, JPF uses the `JPFLogger` class. The tool tracks all classes that call the `JPF.getLogger(String)` API. The **Logger** row in Table 1 shows an example of getting a logger named `IdleFilter`.

- **ChoiceGenerators:** ChoiceGenerators are used to implement new data or thread choices. They are examples of possible extension mechanisms and, therefore, useful for developers of new extensions. The tool tracks methods that register choice generators via the `SystemState.setNextChoiceGenerator()` or `getSystemState.setMandatoryNextChoiceGenerator()` API. The **CG** row in Table 1 shows an example of a CG registration.

- **DirectCallStackFrames:** These are used to implement invocation of synthesized methods (not visible in bytecode). Similar to `ChoiceGenerator`, they are used by developers and together with `ChoiceGenerator`s can lead to potential ill effects on robustness and compatibility of extensions. The tool tracks methods that create `DirectCallStackFrame`s, i.e., its constructor. The **DCSF** row in Table 1 shows an example of a `DirectCallStackFrame` instantiation.

II.1 Static Analysis

The `jpf-autodoc-options` tool performs static analysis at the bytecode level. Using the standard JPF bytecode reader, the tool parses classes under test

Categ	Documentation
Option	et.print_insn - print executed bytecode instructions **defined in:** gov.nasa.jpf.listener.ExecTracker **type:** Boolean **default:** True **used in:** gov.nasa.jpf.listener.ExecTracker **type:** Boolean **default:** False
Logger	gov.nasa.jpf.listener.IdleFilter gov.nasa.jpf.listener.IdleFilter **type:** JPF.getLogger
CG	gov.nasa.jpf.jvm.MJIEnv **type:** setNextChoiceGenerator **method:** setNextChoiceGenerator
DCSF	gov.nasa.jpf.jvm.JVM **method:** pushMainEntry

Table 2: Examples of Generated Documentation

and searches for specific Java bytecode instructions corresponding to each category shown in Table 1. For example, to identify **Options**, the tool searches for `invokevirtual` instructions with the class name attribute `"gov/nasa/jpf/Config"`. The tool searches for 32 different getter APIs from the `Config` class and treats them differently depending on the method signature (e.g., the number and types of parameters to track). The option example in the first row of Table 1 is one of the easiest, with a Boolean value, `iconst_0`, which corresponds to `False`, and the `ldc` instruction with the key name `et.print_insn`.

Config APIs are the most complex to parse; the rest of the categories are parsed in a similar manner: each API is treated based on the bytecode pattern it produces.

II.2 Output Generation

After checking all project files, the obtained data is printed to an XML file using the Castor [6] XML framework. The XML output adds flexibility to the tool, especially if more extensions are to use its output in the future (for example, the `jpf-shell` extension).

Currently, the tool supports generation of wiki pages and uses its own translator to generate wiki files from XML. While generating files, the tool combines the information about the same options and checks for inconsistencies among them. For example, the tool checks for multiple occurrences of the same option and inconsistencies between the options' implementation and their corresponding annotations.

Table 2 shows the documentation generated for the examples in Table 1. The **Option** row shows that the `et.print_insn` option has inconsistencies: its value in the Java source is different from the value defined by its annotation.

The generated documentation is formatted using Google Project Hosting Wiki Syntax [8]. To make wiki easy to view and see potential inconsistencies, the tool employs the color scheme shown in Table 3 for key names. For example, an option highlighted in orange contains inconsistencies between its API call and annotation.

Color	Description
green	The call and annotation are the same
orange	The call and annotation have different values
blue	The call or annotation definition is missing
red	There is more than one call or annotation for a key

Table 3: Keys Color Scheme

III TOOL USAGE

The tool offers a command-line interface, built on top of the Commons CLI library [7]). Table 4 shows the command-line arguments the users can specify. The user can:

Generate an XML file for a project under test:
`jpf-autodoc-options -cp ../jpf-core/build/ -xml`

Generate wiki pages from an XML file:
`jpf-autodoc-options -cp jpf_options_list.xml -wiki`

Generate wiki and XML files:
`jpf-autodoc-options -cp ../jpf-core/build/ -wiki`

Generate wiki and an XML file and save with the -name prefix:

Option	Value	Description
bytecode		view bytecode of classfile
cp	[path]	class path to analyze
dirs		analyze CLASS files
help		print this message
jars		analyze JAR archives
name	[name]	name of project (added as a prefix to generated files)
outdir	[path]	directory to save files
print		print data to console after analysis
test		generate test files
wiki		generate documentation files
xml		generate an XML output file

Table 4: Tool Command-Line Arguments

```
jpf-autodoc-options -cp ../jpf-core/build/ -wiki
-name jpf-core
```

IV EXPERIENCE

The `jpf-autodoc-options` tool has been successfully applied to several JPF extensions: `jpf-core` [5], `jpf-awt` [9] and `jpf-bfs` [10]. The tool found several errors in the `jpf-bfs` project: there were spelling errors in the annotation definitions and a double-definition for one of the option keys. This experience confirms that `jpf-autodoc-options` can be effective in generating project documentation and detecting errors.

IV.1 Limitations

While the tool tries to identify all possible option definitions and implementations, there are limitations to what the tool can do. Because the tool is based on the static analysis techniques, it cannot identify values that are dynamically loaded. In such cases, the tool generates "dynamic" keys and values in the documentation. Such cases should serve as a suggestion to developers to simplify the configuration of their extensions.

V CONCLUSIONS AND FUTURE WORK

We presented `jpf-autodoc-options` [11], a tool for automatic extraction of options and documentation for JPF projects. The tool collects all JPF project options in one place and generates documentation while checking for possible inconsistencies among the options. The tool can be helpful while working with existing JPF extensions, as well as when developing new ones.

In the future, we plan to combine `jpf-autodoc-options` with another similar tool called `jpf-autodoc-types` [12], which extracts information about various types in the JPF projects. Together, both tools can be used to generate JPF project documentation. In addition, we plan to extend `jpf-shell` [13] to work with the `autodoc` tools. Finally, we plan to communicate with the JPF community, which we hope will adapt our tool to document their projects.

VI ACKNOWLEDGMENTS

We would like to thank Google Summer of Code program for sponsoring this project and the JPF team for their constant help with JPF.

References

[1] Java PathFinder Tool-set. http://babelfish.arc.nasa.gov/trac/jpf.

[2] Ariel Rabkin and Randy Katz. Static Extraction of Program Configuration Options. In *ICSE*, 2011.

[3] Configuring JPF. http://babelfish.arc.nasa.gov/trac/jpf/wiki/user/config.

[4] Google Project Hosting http://code.google.com/hosting/.

[5] jpf-core. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-core.

[6] Castor Project. http://www.castor.org.

[7] Apache Commons CLI. http://commons.apache.org/cli.

[8] Google Project Hosting Wiki Syntax. http://code.google.com/p/support/wiki/WikiSyntax.

[9] jpf-awt. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-awt.

[10] jpf-bfs. http://code.google.com/p/jpf-bfs.

[11] jpf-autodoc-options. http://code.google.com/p/jpf-autodoc-options/

[12] jpf-autodoc-types. http://bitbucket.org/carlos_uribe/jpf-automatic-doc.

[13] jpf-shell. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-shell.